A Cluster-on-a-Chip Architecture for High-Throughput Phylogeny Search

Tiffany M. Mintz, *Member*, *IEEE*, and Jason D. Bakos, *Member*, *IEEE*

Abstract—In this paper, we describe an FPGA-based coprocessor architecture that performs a high-throughput branch-and-bound search of the space of phylogenetic trees corresponding to the number of input taxa. Our coprocessor architecture is designed to accelerate maximum-parsimony phylogeny reconstruction for gene-order and sequence data and is amenable to both exhaustive and heuristic tree searches. Our architecture exposes coarse-grain parallelism by dividing the search space among parallel processing elements (PEs) and each PE exposes fine-grain memory parallelism for their lower-bound computation, the kernel computation performed by each PE. Inter-PE communication is performed entirely on-chip. When using this coprocessor for maximum-parsimony reconstruction for gene-order data, our coprocessor achieves a 40X improvement over software in search throughput, corresponding to a 14X end-to-end application improvement when including all communication and systems overheads.

Index Terms—Biology and genetics, distributed systems, parallelism and concurrency, reconfigurable hardware.

1 Introduction

The heterogeneous computing model, where a general purpose CPU is accelerated using a special purpose coprocessor, is a common technique for 3D rendering [1], high-definition video playback [2], and simulation and gaming [3] but has only recently begun to emerge as a widely used, mainstream technique in scientific computing. This is evident by the integration of coprocessor devices into recent high-performance computers such as Los Alamos's Roadrunner, NCSA's Lincoln, Cray's line of XT5 XT5h computers, and SGI's RASC enhancement to their Altix computers. Each of these systems include integrated programmable or reconfigurable coprocessors, specifically IBM PowerXCell processors in the case of Roadrunner [4], NVIDIA GT200-series Tesla processors in the case of Lincoln, and Field Programmable Gate Arrays (FPGAs) coprocessors in the case of Cray [5] and SGI [6].

Although Digital Signal Processors (DSP), Graphics Unit Processors (GPUs), and now so-called Stream Processors are attractive coprocessor devices due to their relatively simple programming model, at this time FPGAs still remain a popular choice for heterogeneous scientific computing. In this case, a special-purpose hardware version of an application's most expensive computation, or *kernel* computation, is implemented in custom FPGA logic. The kernel computations that are traditionally implemented on FPGAs are *O(n)* computations where input data are streamed through a pipeline on the coprocessor. Such implementations are common for numerical linear algebra [7], [8], cryptography [9], pattern matching [10], sequence alignment [11], [12], [13], network intrusion detection [14], and signal, video, and image processing [15], [16], [17], [18], [19]. Note that within the context of application, these kernel computations may be invoked within the higher level algorithm that is not O(n), while the actual coprocessor hardware is limited to O(n) (e.g., sparse matrix-vector multiply performed on the coprocessor and used as a kernel to conjugate gradient method for solving linear systems).

In most cases, kernels implemented on FPGAs are control independent in the sense that their execution behavior does not depend on the input data. They are well suited for large input sets, because despite the fact that they reference each input value only once, their memory access pattern is fixed and known a priori allowing them to hide memory access latency by using a streaming execution model. Their performance is often limited by the FPGA's off-chip memory bandwidth as opposed to the amount of parallelism that can be extracted from the computation [20], [21], [22], [23].

On the other hand, non-*O(n)* kernels, such as those based on quadratic or exponential algorithms, are rarely implemented entirely on the FPGA. Examples of such include many types of optimization and search algorithms. These computations iterate over the same data multiple times, require complex control, exhibit temporal data locality, have unpredictable memory access patterns, and their behavior is dependent on their input data. This class of computation is becoming prevalent in computational biology and includes branch-and-bound and stochastic search algorithms. These algorithms are very well suited to FPGA acceleration because they can take advantage of the extremely high internal memory bandwidth available on FPGAs and thus potentially exploit much fine-grain parallelism as compared to traditional *O(n)* kernels. However, such computations are extremely difficult to implement using traditional logic design tools.

In this paper, we implement the branch-and-bound search algorithm used in direct-optimization phylogenetic reconstruction. In our particular application, phylogenies, which are used to show the possible evolution of species, are represented as unrooted binary trees. The number of possible tree configurations grows exponentially as the number of input species (which correspond to the leaves of the tree)

• *The authors are with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208. E-mail: mintztm@ornl.gov, jbakos@cse.sc.edu.*

Manuscript received 8 Apr. 2009; revised 15 Sept. 2009; accepted 8 Dec. 2009; published online 21 Oct. 2010.
Recommended for acceptance by M. Yamashita.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-04-0159.
Digital Object Identifier no. 10.1109/TPDS.2010.191.

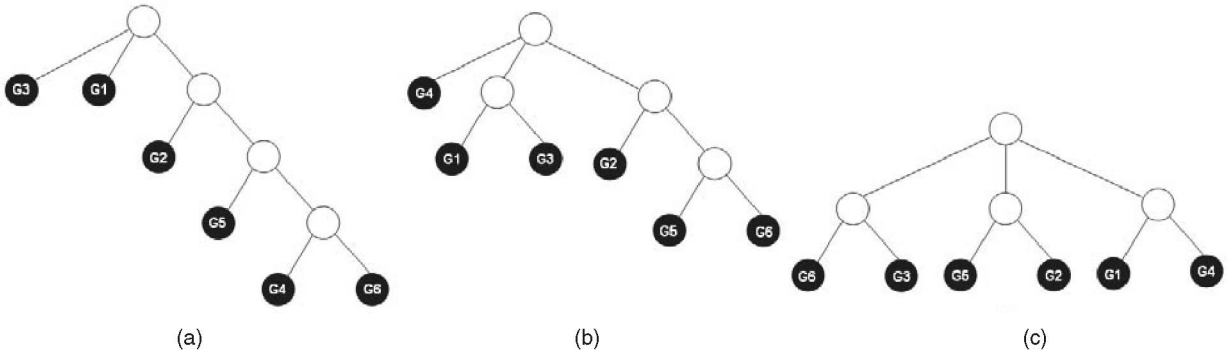

Fig. 1. Three of the 105 possible phylogenies for six input genomes. Input species $(g1, g2, \ldots, g6)$ are shown in black while ancestral species are shown in white.

increases. To search this space, a heuristic or branch-and-bound search method is commonly employed [24]. Heuristic searches often begin with an initial estimate of the tree and makes small branch rearrangements to reach neighboring trees. Popular heuristic phylogeny search methods use simulated annealing [25], [26], [27] and genetic algorithms [28], [29]. More accurate methods rely on a branch-and-bound search, which perform an exhaustive search but discard sets of trees that fail a lower bound computation.

The primary contribution of this paper is an FPGA-based accelerator architecture that is organized as a cluster-on-a-chip, where the coprocessor consists of a set of processing elements (PEs) that each performs the same computation as traditional cluster nodes in the case where the application is run in MPI mode. Our PE design is an FPGA implementation of a kernel that accelerates a combinatorial branch-and-bound search. The search relies on a lower bound computation that is based on pairwise distances. This computation is used in parsimony-based gene-order and sequence phylogeny search, and is compatible with divide-and-conquer heuristics for searching larger tree spaces. Our design extracts both memory-level parallelism and coarse-grain parallelism.

2 Phylogeny Reconstruction

Phylogenetic analysis is the study of evolutionary lineage among a set of species. A phylogeny (or phylogenetic tree) is a binary tree where each vertex represents information associated with a species and each edge represents a series of evolutionary events that effectively transformed one species into another. The analysis of phylogenies is a fundamental tool that biologists use to infer common characteristics across different species based on their evolutionary relatedness. Analysis of phylogenies is a vital component of research in such areas as drug and vaccine development and bio-pathway discovery.

Fig. 1 shows three example phylogenies for six input species. Each of the n leaves has degree 1 and represents a species that currently exists (usually called *taxa*), while each of the $n-2$ internal vertices has degree 3 and represents a species that is an extinct ancestor species. Edge lengths represent evolutionary distance, sometimes vaguely expressed using time (i.e., millions of years) or more precisely using the actual number of evolutionary events from a specific evolutionary model. Both the topology and the edge distances are important characteristics of the phylogeny.

In general, the problem of phylogenetic reconstruction is to approximate through inference the true evolutionary history of n input species. There are several methods that are used for reconstructing phylogenies such as *neighbor joining* [30] and *maximum parsimony* [31], [32], [33]. The benefit of neighbor joining is that its low computational cost allows it to be used for large input sets but generally yields results with poor accuracy. Maximum parsimony methods based on direct optimization are among the most accurate methods but are also among the most expensive [24].

The specific application we target in this work is Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms (GRAPPA) [34], which is an exhaustive search method, moving systematically through the space of all possible phylogenetic trees to find the tree with the lowest sum of edge lengths. For each tree, the program tests a lower bound to determine whether the tree is worth scoring. In practice, such bounding achieves higher than 99.99 percent pruning rates. The lower bound used by GRAPPA is derived from the following theorem, which holds for any graph that obeys the triangle inequality [35]:

Theorem 1. *Let d be a $n \times n$ matrix of pairwise distances between the taxa in a set S; let T be a tree leaf labeled by the taxa in S and w be an edge weighting (tree score) on T, so that we have $w_{ij} = \sum_{e \in p_{ij}} w(e) \geq d_{ij}$, where p_{ij} is a path from i to j on tree T. Set $w(T) = \sum_{e \in E(T)} w(e)$. If $1, 2, \ldots, n$ is a circular ordering of the leaves of T, then we have $2w(T) \geq d_{1,2} + d_{2,3} + \cdots + d_{n,1}$.*

This gives us a lower bound (circular ordering) for the tree score, i.e., the tree score $w(T)$ should at least be $\frac{(d_{1,2}+d_{2,3}+\cdots+d_{n,1})}{2}$. Although this bound is primitive, it is very cheap to compute.

For every tree that is scored, the program will iteratively solve the median problems at internal nodes until convergence, as shown in Algorithm 1 [36].

Algorithm 1. The GRAPPA scoring procedure
Initially label all internal nodes with gene orders
Repeat
  For each internal node v, with neighbors A, B and C, do
  Solve median problem on A, B, C to yield m
  If relabeling v with m improves the tree score, then do it
Until no change occurs

As shown in Fig. 2, the median problem on k genomes is to find a single genome that minimizes the median score

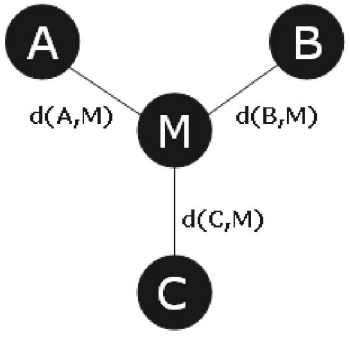

Fig. 2. Given genomes A, B, and C, the median problem is to find genome M that minimizes the median score, where the $\text{median score} = d(A, M) + d(B, M) + d(C, M)$, where $d()$ is an edit distance (i.e., breakpoint distance).

(sum of the pairwise distances) between itself and each of the k given genomes.

For n taxa (and thus n leaves), the number of possible unrooted binary trees is $(2n-5)!! = (2n-5) \times (2n-7) \times \cdots \times 5 \times 3$. To illustrate this growth, assume an arbitrary tree is represented by a circular ordering of its edges that is independent of both the starting point and the direction the edges are read. For example, a tree with three leaves would have three edges and one unique tree based on the circular order; i.e.,

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 1(123) \Leftrightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 1(132) \Leftrightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 2(231) \Leftrightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 2(213) \Leftrightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 3(321) \Leftrightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 3(312).$$

Each of these orderings represent the same tree, i.e., for every ordering edge 1 is connected to edge 2 and 3, edge 2 is connected to edges 1 and 3 and edge 3 is connected to edges 1 and 2. This means that all of the edge orderings will have equal lower bounds per Theorem 1.

The smallest possible unrooted binary tree is a star with three leaves and one internal vertex. For every leaf that is added to a tree, two new edges must be added, which can be inserted by bifurcating any existing edge on the tree. This means that a tree structure with $m+1$ leaves has two possible insertion points greater than a tree with m leaves. A three-leaf tree therefore has three insertion points, for any of these the resulting four-leaf tree will have five insertion points, for of any of these the resulting five-leaf tree will have seven insertion points, etc. Since an n-leaf tree could have been arrived at by $2n-5$ possible paths from a tree with one less leaf, there are $(2n-5)!!$ total tree configurations.

This number grows exponentially as the number of leaves increase. For example, there are only two million trees for 10 taxa, but more than 10^{20} trees for 20 taxa. In general, an exhaustive search for data sets with more than 17 taxa is not feasible [37]. One technique used to remedy this problem is a divide-and-conquer approach where the set of taxa is decomposed into a collection of overlapping subsets where each of which optimizes some criterion designed to make heuristic reconstruction on the subset as accurate and efficient as possible. The best such approach to date is the family of disk-covering methods (DCM) [38], [39]. Tang and Moret combined the DCM1 approach [40] with GRAPPA, limiting the size of the subsets to at most 13 taxa through a combination of threshold choices and recursive calls to the DCM decomposition itself, yielding a DCM-GRAPPA software [41]. The goal of this project was to develop an FPGA-accelerated parallelized tree search architecture, which can perform a high-throughput scan through the tree space. Once achieved, this design will easily be able to be integrated into the DCM approach.

3 Accelerator Architecture

Our tree search architecture provides a simple bijective function for mapping each search state value to a unique tree configuration. Specifically, trees are represented by a list of integers that correspond to the ordering of the tree's edges that define its topology. Tree construction begins with an initial three-edge (four-vertex) tree structure configured in a star topology. The edges are labeled 1, 2, and 3. Each of these edges is an external edge, meaning that one of its endpoints connects to a leaf vertex. The tree is grown by systematically adding vertices to every possible position within the tree, limited to a maximum number of leaves. Whenever a new vertex is added, two new edges are added until a completed tree structure is generated. For each pair of inserted edges, there is an internal edge (both endpoints are internal nodes) and an external edge. The internal edge is always added first, immediately followed by the insertion of the external edge. In our encoding scheme, even numbered edges greater than 3 correspond to internal edges, while edge 2 and all odd numbered edges correspond to external edges.

Figs. 3a, 3b, 3c, 3d, and 3e illustrate the process of tree construction for a tree with seven leaf nodes. Fig. 3a is the initial tree. When internal edges are added, the previous connection between nodes is broken and a new connection is made with this new edge adding both an internal vertex and leaf vertex. As shown in Fig. 3b, internal edge number 4 was added at edge 1, so the connection between the existing endpoint of edge 1 was broken and a new node as well as edge 4 was added to the tree structure. The external edge 5 is then added at edge 4 and its endpoints are connected to the newest nodes. External edges are always added to the internal edge that was inserted immediately before it. This is further illustrated in Figs. 3c, 3d, and 3e. The final tree structure is represented by the edge list: 1 4 5 6 7 2 8 9 3 10 11.

Fig. 4 shows the finite state machine (FSM) controller, which facilitates the sequential process of generating a sequence of trees. In the Insert Edges/LB state, the controller uses various counters to insert the appropriate edges into the appropriate position within the (possibly incomplete) tree on the top of the stack. The tree's lower bound is accumulated in parallel by performing lookups into the distance matrix while each tree is being constructed.

When adding edges, the current tree is logically separated into three parts: the prefix, insertion point, and suffix. The insertion point is the index where new edges are to be added into the tree. The prefix is edges to the left of the insertion point and the suffix is the edges to the right of the insertion point. Before inserting edges into an arbitrary tree, the controller reads the insertion point from the stack.

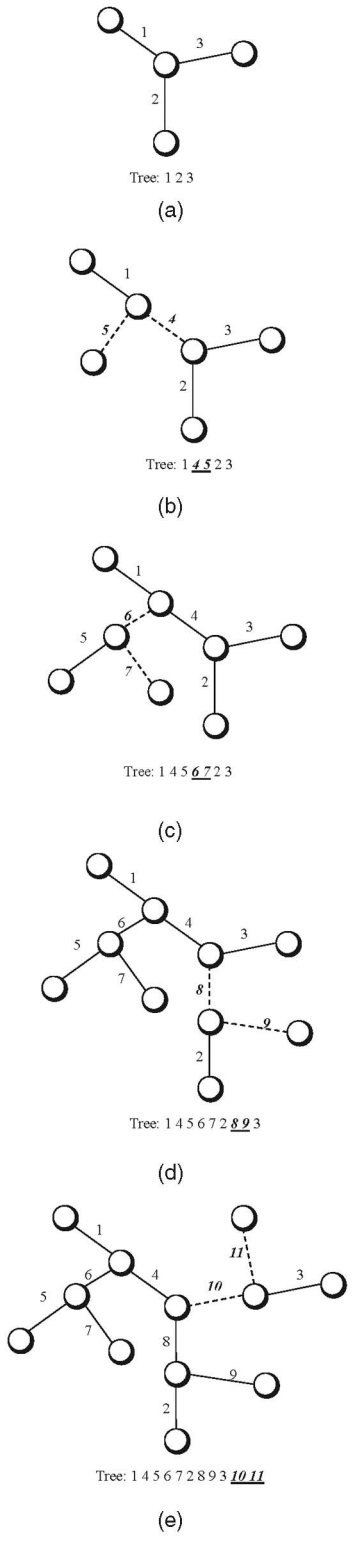

Fig. 3. Stages of tree generation for a tree with seven leaf nodes. (a) Initial stage. (b) Intermediate stage. (c) Intermediate stage. (d) Intermediate stage. (e) Final stage.

Fig. 5 shows the block diagram of the processing element, highlighting its major components. The architecture maintains a stack using on-chip block RAM (BRAM) that is used to store the state of the tree at every stage of construction, implementing a depth-first search where each level of the search tree consists of a tree having an

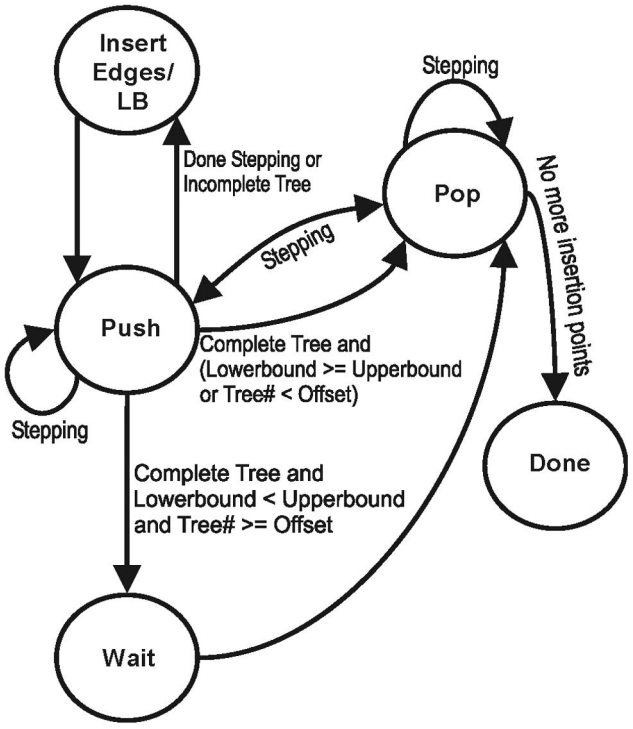

Fig. 4. FSM representation of tree generation.

increasing number of edges. The tree state is striped across four on-chip BRAMs, allowing up to four external edges to be read and written in parallel. Each search state in the stack includes a tree configuration and an insertion point for the next edge to be added to "grow" the tree.

The controller computes the lower bound for every search state, including partial trees in the intermediate stages of tree generation. As each edge is written to the stack, pairwise distances between leaves corresponding to external edges are accumulated in order to keep a running sum of the lower bound value for the current tree. Note that when writing to the stack, four edges are written in parallel

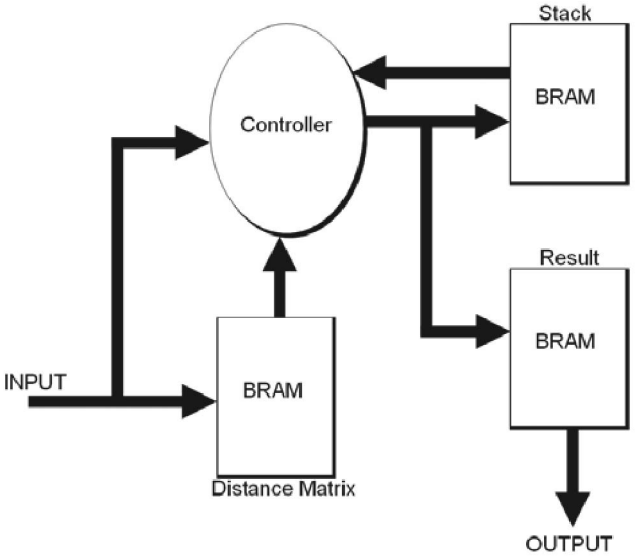

Fig. 5. Black diagram of tree search processing element.

every clock cycle. Since all four edges may potentially be external edges, four copies of the distance matrix are stored in BRAMs within each PE. A PE includes a binary adder tree for accumulating the lower bound value.

As a branch-and-bound search, the PE computes a lower bound value for each tree it generates. Trees are pruned (i.e., skipped over) that have a corresponding lower bound greater than or equal to the parsimony score of the best tree found so far. Despite the fact that the PE computes the lower bound for both partially constructed and fully constructed trees, the PE will only prune fully constructed trees. We made this decision because the potential of pruning incomplete trees greatly increases the logic complexity required to perform tree stepping (described below). This extra logic complexity adds substantial stress to the FPGA's routing network, making it impossible to meet timing closure on the PCI-X interface.

Whenever a full tree is constructed without being pruned, it is considered a candidate tree and the PE stops and holds the tree until it is either read by host or pruned due to a global upper bound update.

3.1 Host-PE Interface

The host interface to the accelerator architecture allows the host to communicate with each PE individually (for setting control registers or retrieving trees whose lower bound is less than the global upper bound) as well as through broadcasts to all PEs (such as for initializing the distance matrices for all PEs or setting the global upper bound).

All PEs are globally initialized with the number of leaves, the initial upper bound value, and the pairwise distances of the input species (computed in software during initialization).

The upper bound value is stored in each PE and determines the pruning behavior during the tree search. Whenever one of the candidate trees found by a PE is scored by the host and its score is less than the current upper bound, the host broadcasts the new upper bound value to all the PEs in order to increase pruning rate.

On the other hand, whenever a PE finds a candidate tree whose lower bound value is less than the global upper bound, the PE suspends its search to allow the host to read the tree. Once the host has read the tree, it sends a continue signal to the PE to force it to resume its search. When a PE is waiting for the continue signal, a new upper bound may be broadcast by the host as a result of it scoring a candidate tree found by another PE. If this new upper bound is less than or equal to the lower bound of the tree that is being held, the PE will discard its current tree and continue searching for the next valid candidate tree without waiting for a continue signal from the host.

We use this polling approach as opposed to interrupts because the host is always in a state where it is either scoring a candidate tree or reading a candidate tree from a PE. As such, there is no reason why the host would need to be interrupted to read a candidate tree. Note that while we assume the host is running only a single thread, this interface model can easily be extended to multiple threads on the host. In this case, one thread can be responsible for reading candidate trees from the PEs and can instance scoring threads as needed.

3.2 Parallelizing the Tree Search

The tree search is "embarrassingly parallel," as the search space can be equally divided across multiple processors.

GRAPPA includes a native cluster execution mode implemented using OpenMPI. When GRAPPA is compiled for cluster execution, the tree generation and bounding procedure is parallelized by using a large set of processors working together to explore the large tree space by equally dividing the tree space into p sections, where p is the step size (number of processors), and k is the offset (processor identifier). Whenever a processor finds a better tree score, it broadcasts this score to the other processors such that they can update their upper bound and increase their pruning rate. This broadcast is the only communication required in this approach; thus, the tree search can achieve near linear speedup.

In this operation mode, processor k will generate tree number $k, k+p, k+2p, \ldots$ We use the same strategy in this project. In this case, p represents the number of PEs on the FPGA(s). In practice, this strategy results in effective load balancing, as in our experiments we did not observe any PE exhaust its search space significantly faster the other PEs in the coprocessor.

Figs. 6 and 7 show how the stepping behavior is implemented. The left side of Fig. 6 shows a snapshot of the tree search where the current tree matches the one shown in Fig. 3. After the PE reaches the bottom of the search tree (and thus has constructed a complete phylogeny), it will pop the stack once, and continue to pop the stack until it reaches a node that has enough bottom-level descendents to accommodate the step size.

As shown in Fig. 7, the PE can readily determine the total number of bottom-level descendents from an arbitrary node by computing

$$\prod_{i=s=2}^{n} (2i-3),$$

where s is the stack size and n is the number of leaves.

Higher step values require a higher average number of stack operations per visit to the bottom level of the search tree than lower step values. Since there is control overhead required for each stack operation, the PE loses some efficiency for larger step sizes. As such, there is a point of diminishing returns when scaling up the number of PEs (and thus the step size for each individual PE).

Implementing the lower bound in hardware also allowed us to unroll the loop used in the lower bound calculation and use replicated copies of the distance matrix memory (stored in on-chip memory banks) to perform multiple iterations of the lower bound loop in parallel. Recall that the lower bound loop looks up and accumulates the circular pairwise distances between each neighboring external edge in the edge ordering of the current tree, using the on-chip distance matrix.

The stepping offset for each core is hard-coded based on the PE's unique identifier. We have implemented up to 20 parallel tree search PEs on our FPGA. This gives stepping offsets ranging from 0 to 19 (i.e., PE 0 has an offset of 0, PE 1 has a offset of $1, \ldots,$ PE 19 has an offset of 19). As we increase the number of PEs instanced on the

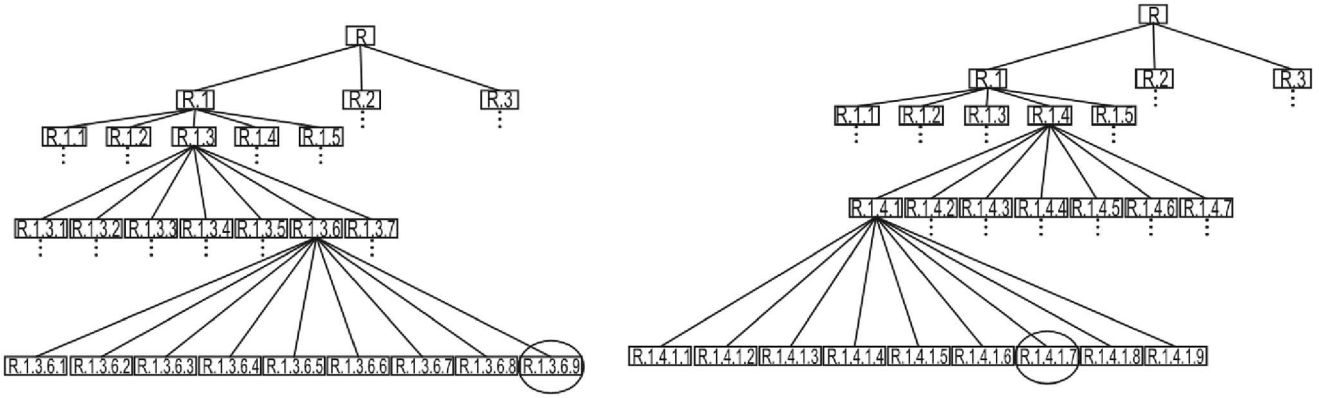

Fig. 6. Graphical depiction of stepping behavior showing the search tree before and after a step of 16 phylogenies. Each node represents a search state and is shown as the stack contents, where "R" represents the root, or three-leaf tree, and the numbers represent the insertion points relative to its parent node.

FPGA, we decrease the number of trees each PE will need to generate and perform the lower bound calculation. While our FPGA has enough logic and memory resources to accommodate 37 PEs, we were unable to achieve a successful place-and-route for more than 20 PEs on our device using Xilinx ISE Design Suite version 10.1. More specifically, coprocessor designs with more than 20 PEs contained very dense, highly connected logic. These mappings produced routing complexities that were too high to achieve timing closure on the fixed PCI-X interface clock, which is a requirement for our FPGA board.

During operation, the host repeatedly scans each of the instanced PEs, reads the waiting candidate tree (if there is one) and its associated lower bound value, sorts the trees in ascending order according to the lower board values, and scores the trees in this order (the intuition is that trees with lower lower bound values are scored first since these trees are more likely to have actual parsimony scores lower than trees with higher lower bounds). Whenever a tree is found to have a lower score than the current upper bound, the new upper bound is broadcast to all the cores and the host will discard any candidate trees having a lower bound that is greater than or equal to the new upper bound.

4 Experimental Setup

In this section, we report our experimental results. The GRAPPA source code, upon which our system is based, is a heavily optimized C program in the sense that memory is carefully managed, the code is written to be extremely efficient, and it employs the most efficient and advanced algorithms of its time.

We created a modified version of the GRAPPA code to serve as the CPU code in our heterogeneous CPU/FPGA system. We replaced the tree search code with programmed input/output calls that serve as "hooks" into the FPGA-based coprocessor. Communication from these program calls was executed using the PCI-X interface. We use the same host for characterizing the performance of both the accelerated and standard versions of the application. Our host machine is a Dell Precision with two 3.06 GHz Intel 2HT Xeon processors.

The FPGA accelerator was design using Mentor HDL Design tools. Our FPGA card is an Annapolis Micro Systems WILDSTAR II Pro card, attached to the host using PCI-X and contains a single Xilinx Vertex-2 Pro 100 FPGA. The FPGA operates at 47 and 40 MHz in the 16-PE and 20-PE implementations, respectively.

4.1 Scalability Analysis

In our first set of tests, we characterized how the performance of the accelerator architecture scales when increasing the number of tree search PEs. As described above, when multiple PEs are used to search a tree space, the space is equally divided among the PEs.

For these tests, we used a set of synthetic data sets, where each data set consists of the extracted leaf genomes from randomly generated trees having 2 to 8 rearrangement events on each edge and whose tree sizes range from 8 to 14 leaves. This corresponds to tree spaces sizes ranging from 10,395 (eight leaves) to 316,234,143,225 (14 leaves).

The goal of these tests is to measure the effective search speed of a single PE and how the search speed scales as we increase the number of PEs. To do this, we set up the test such that no trees are scored by the host by artificially

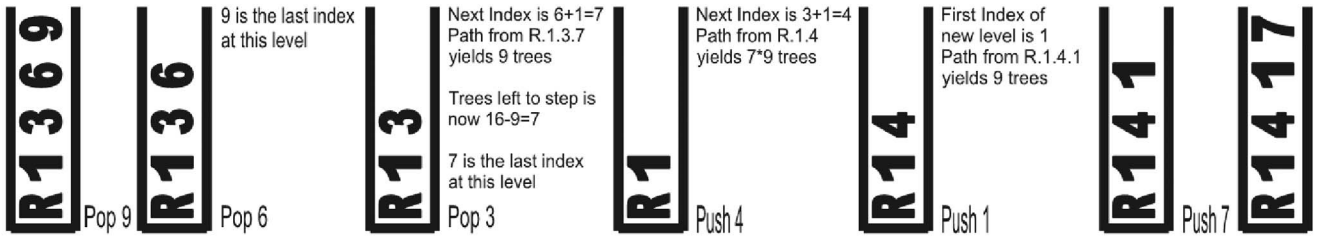

Fig. 7. The stack, as it changes from one search state to the next for a 7-leaf tree when the step size is 16.

TABLE 1
Execution Times in Seconds for 8 to 14 Leaf Tree Spaces Generated Using GRAPPA's Traditional Software and Using the FPGA Parallel Architecture

# Leaves	# Trees	Software	FPGA			
			1 PE	8 PEs	16 PEs	20 PEs
8	1.04E+04	0.06	0.06	0.06	0.06	0.06
9	1.35E+05	0.14	0.11	0.08	0.08	0.08
10	2.03E+06	1.69	0.89	0.30	0.20	0.20
11	3.45E+07	44.00	14.00	2.81	1.95	1.92
12	6.55E+08	898.00	265.00	51.00	34.00	35.00
13	1.37E+10	20442.00	5807.00	1058.00	692.00	727.00
14	3.16E+11	> 604800.00	133172.00	23927.00	15516.00	16255.00

setting the initial upper bound to a value sufficiently low so all trees are pruned by the PEs using the lower bound computation. As such, these tests represent "raw" tree generation and bounding throughput only, and do not include the overheads that would otherwise be required when scoring trees whose lower bound is less than the upper bound. Note that in practice, the ratio of trees that need to be scored (pruning rate) depends on the evolution rate (cluster diameter) of the input set.

For these tests, we compared the time required for a single processing element to search the entire tree space against the time required for 8, 16, and 20 parallel PEs to search the tree space using the stepping technique. Ideally, a parallel multi-PE accelerator would exhibit a linear speedup over a single PE accelerator, but this assumption doesn't consider the overhead required for tree stepping.

For each data set, we also compared the time required for the GRAPPA code to search the same tree space, representing the nonaccelerated, software-only version of the search.

Table 1 shows the execution times of these tests measured in seconds. We found that the software search was unable to complete a search of the space corresponding to a 14-leaf tree within a reasonable amount of time (less than one week of CPU time).

Table 2 summarizes the speedups of the FPGA accelerator implementation over the nonaccelerated implementation, and it also shows the speedup as the number of PEs is increased beyond 1 PE. As shown, a single PE, whose only substantial advantage over software (aside from being implemented in special-purpose hardware) is the fine-grained memory parallelism that it exploits by performing a parallelized lower bound computation, achieves up to 4.5X speedup over software.

The speedup over software of the single PE increases with the sizes of our input data sets (and thus the size of the search space). Because our results include the host-PE communication time required to initialize the coprocessor, longer searches are subjected to less relative overhead.

Further analysis of the table also gives us an optimal number of PEs. Increasing the number of PEs to 20 does not yield the highest speedup in this experiment. There are several factors that contribute to this. One factor is the increased area and routing complexity on the FPGA as the number of PEs is increased. In this case, our 16 PE design operated with a 47 MHz clock rate and our 20 PE design operated with a 40 MHz clock rate. In addition to a lower clock speed, this implementation also has an increased fan-out from the PCI-X interface and consequently required higher pipeline latency for input written to the PEs (in this case, the fan-out was pipelined to decrease routing complexity).

Finally, as described in Section 3.2, the PEs are subject to higher relative control overhead for higher step sizes. Although making the step size larger decreases the number of trees each PE generates, it also increases the number of memory accesses per tree generated for each PE and the number of arithmetic operations per step. This is why there is a point of diminishing returns after 16 PEs.

TABLE 2
Summary of Speedup

# Leaves	# Trees	Speedup over GRAPPA				Multi-PE speedup over 1 PE		
		1 PE	8 PEs	16 PEs	20 PEs	8 PEs	16 PEs	20 PEs
8	1.04E+04	1.00	1.00	1.00	1.00	1.00	1.00	1.00
9	1.35E+05	1.29	1.81	1.81	1.81	1.40	1.40	1.40
10	2.03E+06	1.89	6.75	8.27	8.27	3.56	4.37	4.37
11	3.45E+07	3.14	15.64	22.53	22.89	4.98	7.17	7.28
12	6.55E+08	3.39	17.61	26.41	25.66	5.20	7.79	7.57
13	1.37E+10	3.52	19.32	29.54	28.12	5.49	8.39	7.99
14	3.16E+11	> 4.54	> 25.28	> 38.98	>37.21	5.57	8.58	8.19

TABLE 3
Execution Times for 13 Leaf Data Sets for GRAPPA and the Parallel 16 PE Architecture with Stepping

Input #	Time(sec) GRAPPA	Time(sec) FPGA	FPGA Speedup
1	173207	20793	8.33
2	41930	8856	4.73
3	40838	11207	3.64
4	56203	5554	10.12
5	51453	14121	3.64
6	21641	1578	13.71
7	110273	11161	9.88
8	28181	13168	2.14
9	148571	24114	6.16

When we increase the number of parallel PEs in the accelerator to 16, we achieve a 40X speedup for larger tree spaces.

When comparing the performance of a 16 PE accelerator to a single PE accelerator, the results converged to an 8X improvement. The gap between 8X and the optimal speedup of 16X is also due to the overheads required for the PEs to step through the search space with a step size greater than one. In other words, it takes more cycles for the PE to move to the next tree if it must step over the next n trees as opposed to moving to the next tree according to the search order that is implied by the tree construction algorithm. These results indicate that the PEs require, on average, twice as much time to take steps longer than one when generating trees.

4.2 End-to-End Application Performance

Our second set of tests compare the end-to-end application performance of the unmodified software application, including the scoring of candidate trees whose lower bound is less than the upper bound, to the accelerated application where tree generation and bounding are performed on the FPGA. Note that tree scoring is performed in software by the host in both cases. In these tests, the accelerator consists of 16 PEs, and the input consists of nine unique synthetic, 13-leaf input sets, generated from extracting the leaves of randomly generated phylogenies having 2-8 evolutionary events per edge.

Table 3 summarizes the performance. Even when including the host-FPGA communication overheads, the accelerated version of the application consistently outperformed software, achieving a 3.64X to 13.71X speedup. The variance in speedups is due to differences in both the pruning rate and average scoring time across the input sets.

Input sets that result in higher ratios of scored trees tend to experience lower PE utilization, since the software must cease tree generation when scoring a tree, which can require a significant amount of CPU time. On average, the PEs spent 42 percent of their time searching and 58 percent of their time waiting for their tree to be read by the host.

In the multiple PE implementation of the accelerator architecture, trees are generated in a different order than in the software implementation. This results in a different number of scored candidate trees in the accelerated version of the application.

TABLE 4
Number of Trees Scored by Both Implementations

Input #	# of Trees Scored GRAPPA	# of Trees Scored FPGA	% of Trees Scored GRAPPA	% of Trees Scored FPGA
1	22310765	2912228	0.1623%	0.0212%
2	2772469	1041965	0.0202%	0.0076%
3	2434468	1415052	0.0177%	0.0103%
4	4177687	622079	0.0304%	0.0045%
5	2045558	936178	0.0149%	0.0068%
6	207910	222527	0.0015%	0.0016%
7	7483701	1325416	0.0544%	0.0096%
8	808902	1169870	0.0059%	0.0085%
9	21153803	4031965	0.1539%	0.0293%

Table 4 shows the differences in the number of trees scored and the scoring percentages for each input set. In addition, the time to score a candidate tree is not uniform and can vary greatly depending on the candidate tree itself.

5 Conclusions

We have successfully demonstrated the use of heterogeneous computing with an FPGA-based coprocessor architecture to accelerate the performance of a branch-and-bound computation. The non-$O(n)$ tree generation kernel contains complex control and looping behavior that we implemented entirely on the FPGA, resulting in a 40X speedup for an exhaustive search of a tree space containing hundreds of billions of trees.

We have shown that processing these extremely large data sets is made feasible through this parallelized FPGA architecture that encompasses both fine- and coarse-grained parallelism. By setting the number of processing elements to 16, we obtained our maximum performance. After integrating our accelerator into the GRAPPA algorithm, our parallel architecture achieved an observed 14X end-to-end speedup over the nonaccelerated version of the application.

Acknowledgments

This material is based upon work supported by the US National Science Foundation (NSF) under Grant Nos. CCF-0844951 and CCF-0915608. The authors would like to thank the anonymous reviewers for their helpful comments that enabled them to make significant improvements to this paper.

References

[1] M.P. de Moraes Zamith, E.W.G. Clua, A. Conci, A. Montenegro, P.A. Pagliosa, and L. Valente, "Parallel Processing between GPU and CPU: Concepts in a Game Architecture," *Proc. Computer Graphics, Imaging and Visualisation (CGIV '07)*, pp. 115-120, Aug. 2007.

[2] B. Pieters, D. Van Rijsselbergen, W. De Neve, and R. Van de Walle, "Motion Compensation and Reconstruction of H.264/AVC Video Bitstreams Using the GPU," *Proc. Eighth Int'l Workshop Image Analysis for Multimedia Interactive Services (WIAMIS '07)*, pp. 69-72, June 2007.

[3] "PhysX PPU by Ageia," http://www.ageia.com/pdf/ds_product_overview.pdf, 2011.
[4] "IBM Roadrunner Project," http://www.ibm.com/ibm/ideasfromibm/us/roadrunner/20080609/index.shtml, retrieved, Dec. 2008.
[5] http://www.cray.com, Dec. 2007.
[6] "SGI Products," http://www.sgi.com/products/rasc, Jan. 2009.
[7] J.-W. Jang, S.B. Choi, and V.K. Prasanna, "Energy-and Time-Efficient Matrix Multiplication on FPGAs," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 13, no. 11, pp. 1305-1319, Nov. 2005.
[8] L. Zhuo and V.K. Prasanna, "Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs," *Proc. 18th Int'l Parallel and Distributed Processing Symp.,* p. 92, Apr. 2004.
[9] E. Allen Michalski and D.A. Buell, "The Scalable Architecture for RSA Cryptography on Large FPGAs," *Proc. 16th Int'l Conf. Field Programmable Logic and Applications (FPL '06),* Aug. 2006.
[10] P.D. Michailidis and K.G. Margaritis, "A Programmable Array Processor Architecture for Flexible Approximate String Matching Algorithms," *J. Parallel and Distributed Computing,* vol. 67, no. 2, pp. 131-141, 2007.
[11] A. Boukerche, J.M. Correa, A.C.M.A. de Melo, R.P. Jacobi, and A.F. Rocha, "Reconfigurable Architecture for Biological Sequence Comparison in Reduced Memory Space," *Proc. IEEE Int'l Parallel and Distributed Processing Symp.,* Mar. 2007.
[12] X. Lin, Z. Peiheng, B. Dongbo, F. Shengzhong, and S. Ninghui, "To Accelerate Multiple Sequence Alignment Using FPGAs," *Proc. Eighth Int'l Conf. High-Performance Computing in Asia-Pacific Region (HPCASIA '05),* Nov. 2005.
[13] T. Oliver, B. Schmidt, D. Maskell, D. Nathan, and R. Clemens, "Multiple Sequence Alignment on an FPGA," *Proc. 11th Int'l Conf. Parallel and Distributed Systems—Workshops (ICPADS '05),* July 2005.
[14] Z.K. Baker and V.K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs," *IEEE Trans. Dependable and Secure Computing,* vol. 3, no. 4, pp. 289-300, Oct.-Dec. 2006.
[15] F. Cardells-Tormo and P.-L. Molinet, "Area-Efficient 2-D Shift-Variant Convolvers for FPGA-Based Digital Image Processing," *IEEE Trans. Circuits and Systems II: Express Briefs,* vol. 53, no. 2, pp. 105-109, Feb. 2006.
[16] M. Rawski, P. Tomaszewicz, H. Selvaraj, and T. Luba, "Efficient Implementation of Digital Filters with Use of Advanced Synthesis Methods Targeted FPGA Architectures," *Proc. Eighth Euromicro Conf. Digital System Design,* pp. 460-466, Aug./Sept. 2005.
[17] A. Madanayake, L. Bruton, and C. Comis, "FPGA Architectures for Real-Time 2D/3D FIR/IIR Plane Wave Filters," *Proc. Int'l Symp. Circuits and Systems (ISCAS '04),* vol. 3, pp. 613-616, May 2004.
[18] I.S. Uzun, A. Amira, A. Bouridane, and A., "FPGA Implementations of Fast Fourier Transforms for Real-Time Signal and Image Processing," *IEE Proc. Vision, Image and Signal Processing,* vol. 152, no. 3, pp. 283-296, June 2005.
[19] K.S. Hemmert and K.D. Underwood, "An Analysis of the Double-Precision Floating-Point FFT on FPGAs," *Proc. 13th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* pp. 171-180, Apr. 2005.
[20] K.S. Hemmert and K.D. Underwood, "An Analysis of the Double-Precision Floating-Point FFT on FPGAs," *Proc. 13th Ann. IEEE Symp. Field-Programmable Custom Computing Machines,* 2005.
[21] K.D. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating-Point Performance," *Proc. ACM/SIGDA 12th Int'l Symp. Field Programmable Gate Arrays (FPGA),* pp. 171-180, 2004.
[22] K.D. Underwood and K.S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," *Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* pp. 219-228, 2004.
[23] K.S. Hemmert and K.D. Underwood, "An Analysis of the Double-Precision Floating-Point FFT on FPGAs," *Proc. 13th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* pp. 171-180, 2005.
[24] J. Felsenstein, *Inferring Phylogenies.* Sinauer Assoc., 2004.
[25] A. Stamatakis, "An Efficient Program for Phylogenetic Inference Using Simulated Annealing," *Proc. 19th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS),* Apr. 2005.
[26] J. Zola, D. Trystram, A. Tchernykh, and C. Brizuela, "Parallel Multiple Sequence Alignment with Local Phylogeny Search by Simulated Annealing," *Proc. 19th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS),* Apr. 2006.
[27] D. Barker, "LVB: Parsimony and Simulated Annealing in the Search of Phylogenetic Trees," *Bioinformatics,* vol. 20, pp. 274-275, 2004.
[28] M.J. Brauer, M.T. Holder, L.A. Dries, D.J. Zwickl, P.O. Lewis, and D.M. Hillis, "Genetic Algorithms and Parallel Processing in Maximum-Likelihood Phylogeny Inference," *Molecular Biology and Evolution,* vol. 19, no. 10, pp. 1717-1726, 2002.
[29] A.R. Lemmon and M.C. Milinkovitch, "The Metapopulation Genetic Algorithm: An Efficient Solution for the Problem of Large Phylogeny Estimation," *Proc. Nat'l Academy of Sciences,* vol. 99, no. 16, pp. 10516-10521, 2002.
[30] N. Saitou and N. Nei, "The Neighbor-Joining Method: A New Method for Reconstrucing Phylogenetic Trees," *Molecular Biology and Evolution,* vol. 4, pp. 406-425, 1987.
[31] M. Blanchette, G. Bourque, and D. Sankoff, "Breakpoint Phylogenies," *Proc. Workshop Genome Informatics,* pp. 25-34, S. Miyano and T. Takagi, eds., 1997.
[32] B.M.E. Moret, J. Tang, and T. Warnow, "Reconstructing Phylogenies from Gene-Content and Gene-Order Data," *Math. of Evolution and Phylogeny,* O. Gascuel, ed., pp 321-352, Oxford Univ. Press, 2005.
[33] G. Bourque and P. Pevzner, "Genome-Scale Evolution: Reconstructing Gene Orders in the Ancestral Species," *Genome Research,* vol. 12, pp. 26-36, 2002.
[34] B.M.E. Moret, J. Tang, L. Wang, and T. Warnow, "Steps toward Accurate Reconstructions of Phylogenies from Gene-Order Data," *J. Computer and System Sciences,* vol. 65, no. 3, pp 508-525, Nov. 2002.
[35] B.M.E. Moret, L.-S. Wang, T. Warnow, and S. Wyman, "New Approaches for Reconstructing Phylogenies Based on Gene Order," *Proc. Ninth Conf. Intelligent Systems for Molecular Biology (ISMB '01) in Bioinformatics,* vol. 17, pp. S165-S173, 2001.
[36] B.M.E. Moret, S. Wyman, D.A. Bader, T. Warnow, and M. Yan, "A New Implementation and Detailed Study of Breakpoint Analysis," *Proc. Sixth Pacific Symp. Biocomputing (PSB),* pp. 583-594, 2001.
[37] B.M.E. Moret, D.A. Bader, and T. Warnow, "High-Performance Algorithm Engineering for Computational Phylogenetics," *J. Supercomputing,* vol. 22, pp. 99-111, 2002.
[38] D. Huson, S. Nettles, and T. Warnow, "Disk-Covering, a Fast Converging Method for Phylogenetic Tree Reconstruction," *J. Computational Biology,* vol. 6, no. 3, pp. 369-386, 1999.
[39] U. Roshan, B.M.E. Moret, T.L. Williams, and T. Warnow, "Rec-I-DCM3: A Fast Algorithmic Technique for Reconstructing Large Phylogenetic Trees," *Proc. Third IEEE Computational Systems Bioinformatics Conf. (CSB '04),* pp. 98-109, 2004.
[40] D. Huson, S. Nettles, and T. Warnow, "Disk-Covering, a Fast Converging Method for Phylogenetic Tree Reconstruction," *J. Computational Biology,* vol. 6, no. 3, pp. 369-386, 1999.
[41] J. Tang and B.M.E. Moret, "Scaling up Accurate Phylogenetic Reconstruction from Gene-Order Data," *Proc. 11th Conf. Intelligent Systems for Molecular Biology (ISMB '03) in Bioinformatics,* vol. 19, pp. i305-i312, 2003.

Tiffany M. Mintz received the BS degree in computer engineering from the University of South Carolina in 2003 and the PhD degree in computer science and engineering at the University of South Carolina in 2010. She is currently a Postdoctoral Research Associate at Oak Ridge National Laboratory. Her research interests include computer architecture and parallel, reconfigurable, and heterogeneous computing. She was a GAANN Fellowship recipient from 2003 to 2005, a Pi Fellowship recipient in 2006, a mentor for the US National Science Foundation (NSF) Research Experience for Undergraduates (REU) program in 2006, and a SEAGEP fellow from 2009-2010. She is also a member of the IEEE and the IEEE Women in engineering.

Jason D. Bakos received the BS degree in computer science from Youngstown State University in 1999 and the PhD degree in computer science from the University of Pittsburgh in 2005. He is currently serving as an assistant professor in the Department of Computer Science and Engineering at the University of South Carolina. He has published two dozen refereed publications, was the recipient of the ACM/DAC student design contest awards in 2002 and 2004, and received the US National Science Foundation (NSF) CAREER award in 2009. He is currently serving as information director for *ACM Transactions on Reconfigurable Technology and Systems*. He is a member of the the IEEE, the IEEE Computer Society, and the ACM.